Providing Support for Model Composition in Metamodels

Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, Sudipto Ghosh

Providing Support for Model Composition in Metamodels

Robert France
Department of Computer Science
Colorado State University
Colorado, USA
france@cs.colostate.edu

Franck Fleurey
IRISA
Rennes
France
fleurey@irisa.fr

Raghu Reddy
Software Engineering Department
Rochester Institute of Technology
New York, USA
raghu@se.rit.edu

Benoit Baudry
IRISA
Rennes
France
bbaudry@irisa.fr

Sudipto Ghosh
Department of Computer Science
Colorado State University
Colorado, USA
ghosh@cs.colostate.edu

Abstract

In aspect-oriented modeling (AOM), a design is described using a set of design views. It is sometimes necessary to compose the views to obtain an integrated view that can be analyzed by tools. Analysis can uncover conflicts and interactions that give rise to undesirable emergent behavior. Design models tend to have complex structures and thus manual model composition can be arduous and error-prone. Tools that automate significant parts of model composition are needed if AOM is to gain industrial acceptance.

One way of providing automated support for composing models written in a particular language is to define model composition behavior in the metamodel defining the language. In this paper we show how this can be done by extending the UML metamodel with behavior describing symmetric, signature-based composition of UML model elements. We also describe an implementation of the metamodel that supports systematic composition of UML class models.

1. Introduction

In our earlier work (e.g., see [9]) on aspect-oriented modeling (AOM), a design model consisted of a single primary model and one or more aspect models that each described a feature that crosscuts the dominant structure described in the primary model. Aspect and primary models can be composed to obtain an integrated design model that can be analyzed by existing tools to uncover conflicts and undesirable emergent behavior. It has become apparent that the AOM approach we developed supports a more general form of separation of concerns in which a design is described by a collection of design views. Here, a view is a model that describes how a particular subset of design concerns are addressed. One can still classify the views as primary and aspect, but this distinction is not necessary when using the symmetric composition techniques we developed for composing AOM models [18].

Model composition involves navigating through models and manipulating structures of model elements. Non-trivial models expressed in languages such as the Unified Modeling Language (UML) [21] have complex structures and thus manual composition can be tedious and error-prone. Tools that provide significant automated support for model composition are needed to reduce the accidental complexities associated with manually composing models.

In a previous paper we described a composition metamodel that defined basic model composition behavior [18]. The composition metamodel is an extension of the UML metamodel that includes behavior supporting symmetric, signature-based composition of model elements. In this paper we describe an extension of the composition metamodel that implements flexible model composition through the use of *composition directives* [18]. We also describe the implementation of the metamodel in the Kermeta language [15, 22].

The remainder of the paper is structured as follows. In Section 2 we give an overview of signature-based class model composition. A basic composition metamodel for signature-based composition is described in Section 3. An extended composition metamodel that provides support for tailoring the composition is presented in Section 4. In Section 5 we describe the limitations of the current metamodel

and its implementation, and we discuss insights gained while developing the metamodel. Related work is discussed in Section 6 and we give our conclusions in Section 7.

2. Symmetric Signature-Based Model Composition

The composition of design views can be structured into two major phases:

1. **Matching Phase:** In this phase, model elements that describe different views of the same concept are identified. These elements will be merged in the *Merging Phase* to obtain an integrated view of the concept.
2. **Merging Phase:** In this phase, matched model elements are merged to create new model elements that present integrated views of design concepts.

2.1. Matching Model Elements

The process of identifying model elements that describe different views of the same concept is called *element matching*. Matched elements are merged to form a single element that provides an integrated view of the concept. To support automated element matching, each element type is associated with a signature type that determines the uniqueness of elements in the type space[1]: Two elements with equivalent signatures cannot coexist in a model.

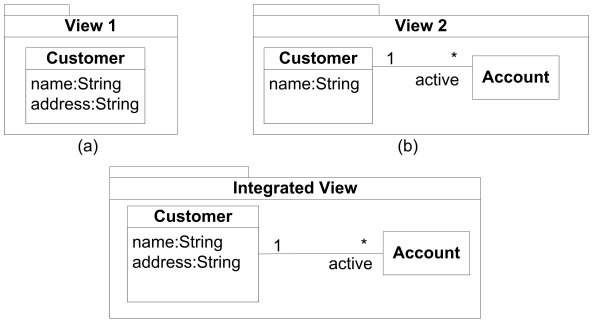
A signature type is a set of syntactic properties associated with an element type. A model element's signature consists of the values associated with these properties. For example, if the signature type of a UML class consists of the properties *name* and *isAbstract*, then the signature of a concrete class with name $ConcClass$ is $\{name = ConcClass, isAbstract = false\}$. If two views each have a class with the same name and with the same value for $isAbstract$ then the two classes match and are merged during view composition to form a single class in the composed model. The merged class contains the union of the attributes and operations in the source classes (syntactically equivalent operations and attributes are included only once in the merged class).

A signature type that consists of all syntactic properties associated with a model element is said to be *complete*. If an element type is associated with a complete signature type then elements of the type match if and only if they are syntactically equivalent. Complete signature types are typically used for matching model elements that must be syntactically identical across the views (i.e., different views of these concepts are not allowed). One can consider these concepts as the primitive building blocks of the problem concept space. An example of a UML model element type that can have a complete signature type is $Property$ (instances of this type include class attributes).

Classifiers and other container elements that are used to present different views of the same concept are associated with signature types that are not complete. For example, the signature type for a class does not include its attributes and operations because these are used to present different views of the same concept (i.e., different views of the same concept have different attributes, associations, or operations).

2.2. Simple Merging of Model Elements

An example of signature-based merging of class models is given in Fig. 1. The figure shows parts of two views: *View1* describes the customer concept from a marketing perspective, while *View2* describes the customer concept from an account management perspective. The two views to be merged include class, attribute, and association elements and thus signature types must be associated with these elements before the models can be composed. In this example, the signature type for a class consists only of its name property, while the signature types for the other elements are complete. The $Customer$ classes in *View 1* and *View 2* match because they have the same name. The $name$ attributes in the $Customer$ class views are syntactically equivalent and thus $name : String$ is included once in the merged class. The class $Account$ appears in one view and not in the other and thus it is included in the composed view. The result is the composed view shown in Fig. 1(c).



(c) Merging using a class signature type consisting only of class names.

Figure 1. An Example of Model Element Matching and Merging

Default merge rules are needed in the cases where the elements to be merged have different values for a single-valued property that is not included in the signature. Consider the case in which a view consists of an abstract class with the same name as a concrete class in another view. These classes will be matched in a composition of the views

[1] A type space is the set of instances of a class (type) in the metamodel

that uses a class signature type consisting only of the class name property. Merging these classes is not possible unless there is a rule that determines the value of the single-valued $isAbstract$ property associated with the merged class in the integrated view. The default rule in this case is that the merged class will be abstract. Modelers can override these default merge rules if they are deemed not appropriate.

Merge rules are also needed when different constraints are associated with matching model elements. For example, if two matching attributes are associated with different invariants then a rule is needed to determine how the constraints are to be combined to form a constraint associated with the merged attribute. In this case the default merge rule will associate the conjunction of the invariants with the merged attribute. Similarly, if matching associations have different multiplicities at their corresponding ends, then the default merge rule associates the weaker multiplicity with the end of the merged association in the composed view.

2.3. Flexible Model Merging with Composition Directives

Signature-based merging of models can be fully automated, but its use can lead to matching of model elements that represent different concepts (concept mis-identification) and that fail to match elements with different signatures that represent the same concept (concept misses). Two models are said to be incompatible with respect to composition if merging yields a model that does not accurately describe the design concepts. Currently, the AOM approach does not provide an automated means for determining model compatibility before models are merged. Developers have to use human judgment or they can compose the views and analyze the composed model to determine if the models are compatible. Providing automated support for determining model compatibility is the focus of our ongoing work on the next generation of AOM techniques. In the remainder of this subsection we outline some situations that can give rise to incompatible models and describe how composition directives can be used to enhance the compatibility of models once the sources of incompatibilities are identified.

The names associated with model elements are often key to finding matching concepts. Consequently, signature types associated with named model elements typically include the name property. The use of names to determine matching elements relies on modelers using names consistently in their models. Inconsistent use of element names is likely to occur in situations where the models to be composed are developed by different modeling teams. For example, a model may have a class named $Customer$ that represents the concept of a customer, while another model may represent the same concept using a class named $Client$. If a class signature type that includes the name property is associated with classes, then the signature-based approach will fail to match these classes and they will be included in the composed model as representations of different concepts. This is an example of a concept miss.

As an example of concept mis-identification, consider the case in which we have a view created from a billing perspective that consists of a class named $Client$ representing a paying customer, and another view in the same problem space but from the perspective of a supply-chain perspective that consists of a class named $Client$ representing a supplier. These classes will be merged under a name-based signature type resulting in concept mis-identification.

The above naming problems are just one of many types of model incompatibility problems that lead to faulty compositions. A more serious kind of incompatibility problem arises when different types of constructs are used to represent the same concepts across views. For example, a concept may be represented by a class in one view and by an attribute in another.

While our composition approach does not currently support automatic detection of model incompatibilities, it provides mechanisms that modelers can use to resolve some of these problems once they have been identified by the modeler. Modelers can specify *composition directives* that are used during composition to force matches, disallow merges, and to override default merge rules. Two types of composition directives are currently supported in the composition metamodel described in this paper:

Pre-Merge Directives : These directives specify simple model modifications that are to be made before the models are merged. These changes will force or disallow element matches. For example, in the case where a view uses a $Customer$ class and another view uses a $Client$ class to represent the same concept, a $Rename$ pre-merge directive can be applied in one view to change the name of the class so that it matches the name in the other view. Similarly, if two views use classes with the same name to describe different concepts (as in the billing and supply-chain example given earlier) a $Rename$ pre-merge directive can be used to change the name in one view so that the classes differ in their signatures and thus are not matched during the merge phase.

Post-Merge Directives : These directives specify simple modifications to the merged model. For example, it may be the case that a security view requires the removal of associations that are present in other views. This restriction can be specified as post-merge directives that remove these associations from the merged model.

The following are the types of modifications that can be

specified using pre-merge and post-merge directives:

Create : Creates a new model element

Remove : Removes a model element from a namespace in a model

Add : Adds a model element to a namespace in a model

Set : Assigns a value to an element property

A pre-merge directive is applied to the views to be merged, while a post-merge directive is applied to a view resulting from merging different views. The modifications are intended to be refactorings, that is, they should not change the essential behavior of the models they are applied to. For example, one should not use a composition directive to remove features that provide required services nor to add features that provide new services to users. Composition directives should only be used to restructure the models so that the features they define can be integrated. We currently do not have mechanisms that enforce proper use of composition directives and thus it is the responsibility of the modeler to ensure that composition directives are appropriately applied.

View merging with directives is structured into three phases. In the *Pre-Merge* phase pre-merge directives are used to refactor the views before they are merged. In the *Merge* phase, the refactored views are merged using signature types to identify matching elements and rules to merge matched elements. The result of this phase is called a merged model. In the *Post-Merge* phase, the merged model is refactored to produce the integrated view. For example, a post-merge directive can be used to add a relationship between a model element introduced by one view and an element introduced by another.

3 A Metamodel for Signature-Based Composition

In this section we describe how the UML metamodel is extended to support signature-based composition of model elements. We also describe an implementation of the metamodel that provides automated support for composing class models. The metamodel is implemented using the Kermeta Language [15, 22].

The metamodel is presented in two parts. This section describes the UML metamodel extensions that support the merging of model elements using signatures and merge rules. Section 4 describes the extensions used to support the use of pre-merge and post-merge directives.

3.1 Merging Models

Fig. 2 shows the core metaclasses and operations that are added to the UML metamodel to support merging of models. The new metaclasses and relationships are highlighted. The metamodel shows only a subset of the affected metaclasses in the UML metamodel: $Element$, $Classifier$, $Property$ and $Model$.

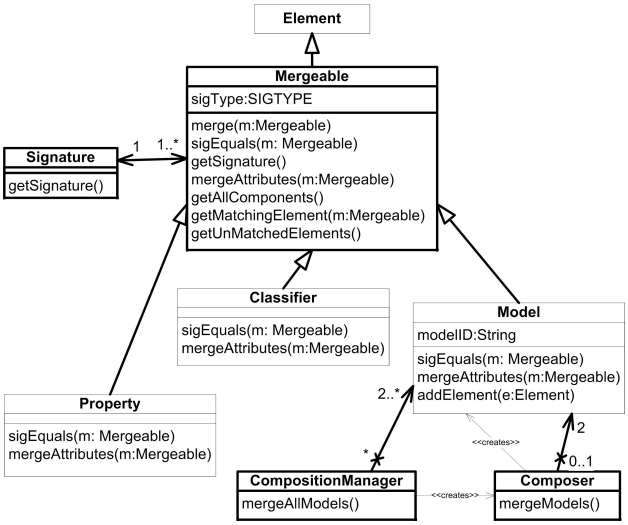


Figure 2. Merge View of Composition Metamodel

The metaclass $Mergeable$ represents the UML model elements that contains operations that can be used to merge them with elements of the same type. The $Mergeable$ metaclass has a meta-attribute $sigType$ that specifies its signature type. The signature type determines the instance of $Signature$ that is associated with an instance of $Mergeable$.

The $Model$ metaclass is extended with an attribute and an operation. The $modelID$ meta-attribute is used to identify models that belong to the same aspect-oriented model: Views that have the same $modelID$ belong to the same aspect-oriented design model. The operation $addElement$ is used to add elements to models.

An instance of the $Composer$ metaclass coordinates the merging of two models. It has an operation $mergeModels()$ that checks the model signatures of two views to be merged and merges the views to produce a new model when the signatures match. The $CompositionManager$ metaclass coordinates the merging of all views in an AOM design model. The $mergeAllModels()$ operation creates a $Composer$ instance that is associated with two of the views to be merged and then calls the $mergeModels()$ operation. After the composed view is produced, the $mergeAllModels()$ operation then associates the composed view and another view with the $Composer$ instance and invokes the $mergeModels()$ operation. This process continues until

all the views are composed.

An instantiation of the composition metamodel includes a set of views (models), in which each model element is associated with a signature class that is used to obtain the signature of the model element. An instantiation also consists of an instance of the $CompositionManager$ metaclass that is responsible for merging of the views.

The sequence models shown in Fig. 3 describe a composition scenario in which two models are successfully merged. In the scenario, a $Composer$ instance that is linked to a model, pm and another model am has been created by the $Compositionmanager$ instance (this creation is not shown). The scenario starts with the invocation of the $mergeModels()$ method in the $Composer$ instance as shown in Fig. 3(a). The $Composer$ instance then checks whether the signatures of the two models to be composed are the same. The signature of a model consists only of its $modelID$. If the model signatures match then pm is requested to merge itself with am. If the model signatures are not the same then the models are not composed.

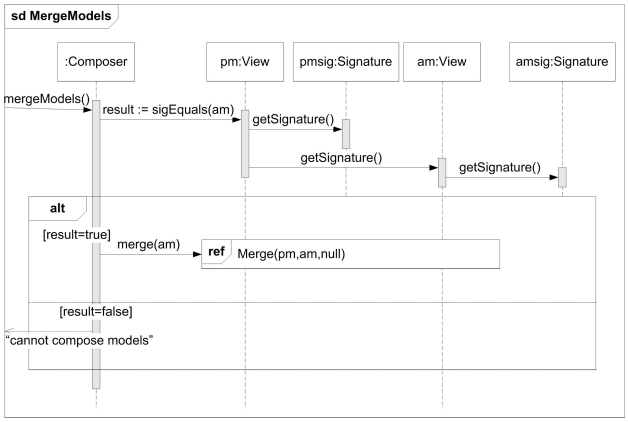


Figure 3. Sequence Model for Model Composition Scenario

The merging of two models is described by the *Merge* interaction fragment. This fragment recursively composes models and their constituent elements and creates a new composed model.

3.2 Implementation with Kermeta

This section details how the merging of models proposed in the previous section was implemented using Kermeta. Kermeta [15, 22] is an open-source metamodelling language developed by the Triskell team at IRISA. It has been designed to be a common basis for implementing metadata languages, action languages, constraint languages and transformation language [22]. It can be used to define the structure and behavior of a user-designed metamodel.

The Kermeta metamodel is divided into two packages: $structure$ and $behavior$. The $structure$ package corresponds to the OMG metamodelling language Essential Meta-Object Facility (EMOF) [14]. The $behavior$ package corresponds to a statically typed action language that is used to define the behavior of metamodels.

The Kermeta action language includes object-oriented features and model-specific features. Some of these model-specific features, such as the handling of first-class association and composition, are used in the implementation of the model composition algorithm. In addition, Kermeta implements OCL-like closures such as $each$, $collect$, and $select$. Inclusion of these features makes it easier to implement operations defined in the metamodels.

The Kermeta language was chosen to implement the composition metamodel for the following three reasons:

- Kermeta allows one to implement operations defined in the composition metamodel.
- Kermeta includes reflection capabilities that allow for a generic implementation of the composition algorithm.
- Kermeta tools are compatible with the Eclipse Modeling Framework (EMF) and thus Eclipse tools can be used to edit, store, and visualize models manipulated in the AOM approach.

The Kermeta implementation of the $merge$ operation, which is responsible for the composition of two objects, (see Fig. 2) is defined generically and uses reflection to determine the specific type of object to compose. The result of this approach is that the implementation of the merge operation in class $Mergeable$ is used to merge all types of UML elements. As Kermeta was defined as an extension of the EMOF standard, the reflection capabilities of Kermeta were inherited from EMOF. The EMOF reflection classes used in the implementation of the composition metamodel are shown in Fig. 4.

All EMOF classes inherit properties from the $Object$ class. This class contains the following operations that are used in the Kermeta implementation of model composition behavior:

- The $getMetaClass()$ operation returns the $Class$ of an object. For example, if the $getMetaClass()$ is used on an operation, it will return the metaclass $Operation$.
- The $container()$ operation returns the containing parent object.

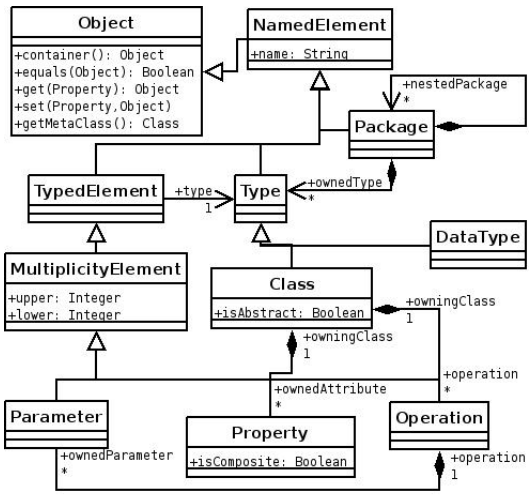


Figure 4. EMOF Classes Used in the Composition Technique

- The $equals(element)$ determines if the element (an instance of $Element$ class) is equal to the instance in which the operation is invoked.

- The $set(property, element)$ operation sets the value of the property to the element.

- The $get(property)$ operation returns a list or a single value depending on the multiplicity.

The $isComposite$ attribute defined in the class $Property$ returns true if the object is contained in the parent object. Cyclic containment is not possible, i.e. an object can be contained in only one other object. The attributes, $upper$ and $lower$, of class $MultiplicityElement$, represent the multiplicities of the associations at the metamodel level. For example, "0..1" represents a lower bound "0" and an upper bound "1".

Additionally, the $getAllProperties()$ operation is added to the $Object$ class. It returns all the properties (including inherited properties) associated with the object instance. This will return elements that are composite as well as primitive. The primitive elements shown in Fig. 4 are String, Boolean, Integer datatypes.

Fig. 5 presents a partial listing of the $merge$ operation in the class $Mergeable$. In this listing the $getMetaClass$ operation defined in EMOF is used to check the types of the objects to merge and to instantiate the merged object. The $getAllProperties$ operations is used to iterate on all properties of the objects to be merged.

4 Using Composition Directives

This section presents the language we use for composition directives. The abstract syntax of the language is specified by a set of classes in the composition metamodel. Fig. 6 shows the part of the composition metamodel that provides support for the use of pre-merge and post-merge directives. We defined a textual concrete syntax for each type of directive. To support the execution of directives we implemented Kermeta operations in the metamodel that provide operational semantics for the directives.

4.1 The Directives Metamodel

An instance of $Composer$ can be associated with two types of directives: $Create$ and $Change$ directives. $Create$ directives are used to create new model elements. A newly created element is not associated with a namespace, and thus must be added to a namespace during composition. $Change$ directives are used to modify model elements. These directives can be used to remove an element from a namespace, set a property value associated with an element, and add an element to a namespace.

A $Change$ directive is associated with a reference to the model element it modifies. A Set directive is associated with two instances of $ElementRef$; one is the target property and the other is the new value for the property.

Elements can be referenced by (1) a name that is an instance of $NameRef$, (2) their literal value, or (3) a unique identifier that is an instance of $IDref$.

The refactorings defined by the pre-merge and post-merge directives are accomplished by invoking the $execute()$ operation in each directive. The directives are

```
operation merge(other : Mergeable)
                      : Mergeable
  pre same_type is
        self.getMetaClass
            == other.getMetaClass
  is do
    // Create the merged object
    result :=
       self.getMetaClass.new.
                  asType(Mergeable)
    // iterate on all properties of
    // the objects to merge
    var properties :
       Collection<Property> init
             self.getAllProperties()
    properties.each{ p |
      if Mergeable.isSubType(p.type)
        then
        // The merge operation is
        // recursively called for
        // the values of this
        // property to compute
        // the value of the
        // property for the merged
        // object
        [...]
result.set(p, merged_value)
      else
        // The value of the property
        // for the merged object
        // is computed from the
        // value of the property for
        // the objects to merge.
        // Some conflicts can be raised
        // if the multiplicity of the
        // property is 1 and the objects
        // to merge
        // have different values.
        [...]
result.set(p, result_value)
      end
    }
  end
```

Figure 5. Excerpt From the Kermeta Implementation of Merge

executed in the order they are presented by the modeler. During execution of $Change$ directives, the element references are used to gain access to the model elements to be modified. Execution of a $Create$ directive results in the creation of a new model element. A $Create$ directive should be followed by an Add directive that places the newly created element into a namespace.

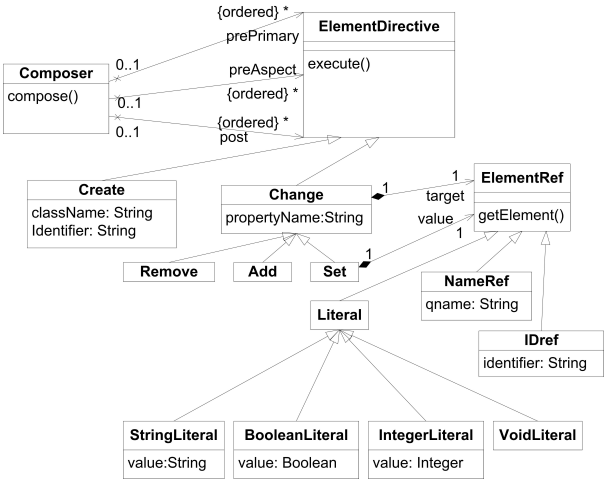


Figure 6. Directives View of Composition Metamodel

4.2 Implementation of Composition Directives

The implementation of composition directives is designed around a command design pattern [10]. For each concrete sub-class of class $ElementDirective$, the abstract operation $execute()$ is implemented in Kermeta. An execution context element is introduced in the implementation of the composition metamodel to store the bindings between identifiers and model elements. The $execute()$ operation in class $Create$ first searches for the class to instantiate from its name, then the class is instantiated and the object is associated with the identifier in the context.

The change directives are used to change the value of the object properties, which can be attributes or associations in EMOF. The $Change$ class is abstract and the $execute()$ operation has to be defined in its sub-classes. If the multiplicity of a property is 1 then the set directive can be used. For instance, a set directive can be used to change the value of the property $name$ of a class. If the multiplicity of a property is greater than one, then add and remove directives have to be used. For instance, the add or remove directive must be used to change the value of the property $ownedAttributes$ of a class, that is, to add or remove attributes from a class.

During the execution of directives, references to actual objects must be resolved (class $ElementRef$). The abstract operation $getElement()$ defined in class

$ElementRef$ serves this purpose and is defined in every sub-class. For literals the object corresponding to the value is instantiated and returned. For name references a lookup in the model is performed to retrieve the corresponding object. For instances of the $IDRef$ class, the object is resolved from the context associated with the composition directive.

When directives are defined, the models that they are to be applied to are not necessarily available. This makes it difficult (if not impossible) in some cases to check that a directive makes references to existing objects, to existing properties or uses the appropriate type of values for a property. These kind of errors have to be handled during the execution of directives. In the Kermeta implementation, we have defined different types of exceptions corresponding to these errors. These exception are raised when the $execute()$ operations encounter an error.

For demonstration purposes, we use a simple concrete syntax to specify the location of models, and to express the directives that are to be applied before and after merging. A more comprehensive syntax for composition directives was introduced in a previous paper [18].

Fig. 8 shows directives that are applied to the input model shown in Fig. 7. The first two directives illustrate the use of the set directive to change the name of class B to C and to change the role name of the association between classes A and B to c. The third directive creates a new class and associates it to the identifier e. In the concrete syntax, identifiers are preceded by the symbol \$. The fourth directive sets the name of this new class to E. The next directive adds class E to the package P. The last two directives remove the datatype D from package P and remove the attribute d of type D in class A. The output model obtained by applying these directives is presented on Fig. 7. Note that the order in which directives are applied is specified by the order in which they appear in the textual description.

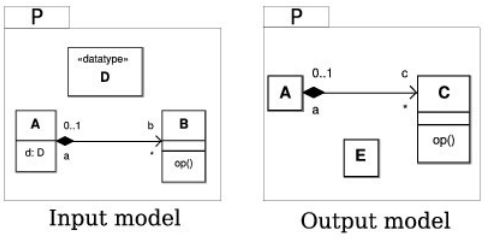


Figure 7. Example Input and Output Models

Another small example of the use of composition directives is given in Fig. 11. In this example a model describing a Bell-LaPadula (BLP) access control feature [3] (see Fig. 10) is to be composed with a model of a

```
// Set directive examples
P::B.name = "C"
P::A::b.name = "c"

// Create directive examples
create Class as $e

// Set the properties of the new objects
$e.name = "E"

// Add directive example
P.eClassifiers + $e

// Remove directive example
P.eClassifiers - P::D
P::A.eStructuralFeatures - P::A::d
```

Figure 8. Example Directives

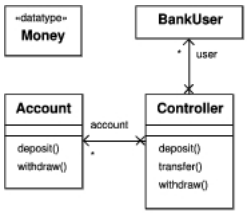


Figure 9. Banking Application Model

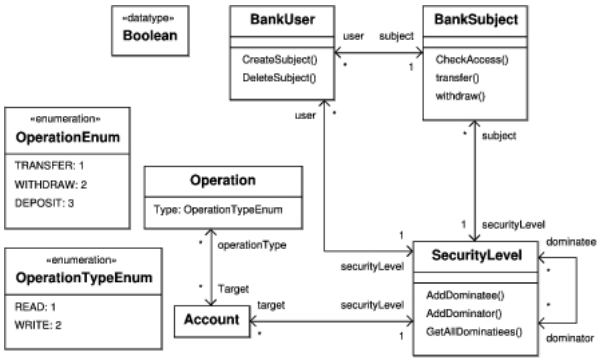


Figure 10. BLP Security Model

banking application (see Fig. 9). In BLP, users (e.g., $BankUser$ instances) and objects under access control (e.g., $Account$ instances) are each associated with a security level. $BankSubject$ defines the banking services that are invoked by users with security levels. A dominance relationship is defined among security levels and is used to determine the type of access a user has to an object. For example, a bank user has read access to an account only if

the user's security level dominates the security level of the account.

In this composition example, the model name is used to match models and thus a directive is needed to change one of the input model names so that they can be matched[2]. The transfer and withdraw operations have been put under access control and thus they cannot be present in the $Controller$ class in the merged model. This is specified as a post-merge directive.

The user specifies in a text file the URIs for the two input models (PM and AM precede the URIs of the models), the URI for the resulting model (preceded by CM) and the composition directives. The $PMPre$ and $AMPre$ sections specify pre-directives on the models. The $Post$ section is used to specify post-directives to be applied on the merged model. The file is parsed to produce an instance of the composition metamodel. The instance is then loaded into the Kermeta environment and executed following the process described earlier. The result of the composition is displayed in Fig. 12.

```
// PM Model URI
PM "Bank.ecore"

// AM Model URI
AM "BLP.ecore"

// Composed Model URI
CM "BankBLP.ecore"

// predirectives for PM model
PMPre { }

// predirectives for AM model
AMPre {
// Rename package BLP to Bank
BLP.name = "Bank"
}

// postdirectives
Post {
  Bank::Controller.eOperations
      - Bank::Controller::transfer
  Bank::Controller.eOperations
      - Bank::Controller::withdraw
}
```

Figure 11. Textual Input for Composing the Bank Model with the BLP Model

[2]The $modelID$ meta-attribute has not yet been implemented in Kermeta

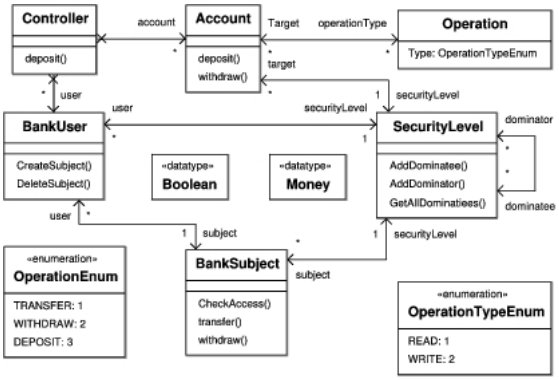


Figure 12. Composed Model

5. Discussion and Future Work

We leveraged the ability of Kermeta to quickly produce implementations of composition metamodels when evolving the early versions of the composition metamodel. We were able to identify errors in our metamodel sooner and thus we were able to converge, in a very timely manner, on a technically sound composition metamodel. Before we developed the Kermeta implementation we were working on implementing a signature-based composition mechanism in Java. This work continued while we were developing the Kermeta implementation, but we were able to complete the Kermeta implementation sooner. Some of the difficulties that the Java implementers had were related to using the mostly undocumented API's for manipulating UML models in Eclipse[3] and to manipulating large XMI representations of the models. The Kermeta environment shields some of this accidental complexity from its users.

The current Kermeta implementation of the composition metamodel is a demonstration-of-concept prototype, but its use shows that Kermeta is capable of providing the support needed to compose and, in general, manipulate models. The current implementation provides a good demonstration of the role that metamodels can play in developing automated support for model composition, but it has its limitations. In the remainder of this section we give an overview of the limitations and outline our plans to address these limitations.

5.1. Generating the Composition Infrastructure

An AOM environment that effectively utilizes the metamodel for signature-based composition should provide at least:

[3]http://www.eclipse.org

- support for building design views that conform to the metamodel,
- a mechanism that generates an instantiation of the composition metamodel given a non-empty set of design views, and a set of composition directives, and
- a mechanism for invoking the model composition behavior in a conforming model.

In such an environment the modeler is responsible for developing the design views and composition directives, while the environment is responsible for producing the infrastructure needed to compose the models. To discharge its responsibility, the environment requires (1) a mechanism for creating and processing composition directives to produce composition infrastructure elements that implement the directives, and (2) a mechanism for defining and processing signature types to produce composition infrastructure elements that extract model element signatures when needed during composition.

The Kermeta implementation provides support for all of the above, but with limitations.

Generating the composition infrastructure currently requires one to have intimate knowledge of the Kermeta environment. We plan to develop a model composition interface that removes the current accidental complexities associated with generating and using composition infrastructures within Kermeta. The Kermeta environment is based on EMOF and has been embedded in an Eclipse modeling environment. An existing Eclipse modeling tool is currently used as a frontend for creating models. The interface we plan to develop will extend the frontend so that it provides an interface for generating the composition infrastructure needed to compose the models.

Another limitation of the current Kermeta implementation is that the signature types associated with meregeable model elements are fixed. We are currently developing support for specifying and processing signature types. In the approach that we are developing the current fixed signature types will become default signature types that are used if the modeler does not specify a signature type. We will provide an interface that would allow a modeler to change the signature type of a model element. To generate the signature part of the composition infrastructure, the environment will use the signature type to produce an instance of the $Signature$ metaclass that retrieves the values of the model element's meta-attributes that make up the signature using model reflection mechanisms.

5.2. Composing UML Models

The composition metamodel currently supports only class model composition. In principle the signature-based approach is applicable to any UML model element. Applying it to behavioral models involves determining the elements that are mergeable, and developing merge strategies and rules for the elements that are to be merged. Consider the case of merging sequence models. One first has to determine how to decompose a sequence model in order to support systematic processing of its components. If one attempts to decompose a diagram by lifelines then the problem of how to handle interaction fragments that span lifelines is raised. Decomposing a sequence model in terms of its fragments may be better. One can treat fragments and their constituent lifelines as mergeable elements. The rules for merging fragments and their constituent lifelines would be more intricate than the rules for merging class model elements primarily because there are many possible ways in which the events in a fragment can be combined with other events. We are currently exploring ways in which the signature-based approach can be extended to sequence and other UML models.

If each design view consists of more than one model type (e.g., a view consisting of both class and sequence models) then one has to be additionally concerned with ensuring that composition produces an integrated view that consists of a consistent collection of models. One way of doing this is to transform the models in one view to a single form that contains all the information in the different models. This single form can then be used as the source model in a merge. In the case of the UML, this single form could be an XML or XMI representation of an instantiation of the UML metamodel.

The composition approach we described in this paper is based on syntactic properties. We are currently developing a next generation of AOM techniques that will support the use of composition operators that preserve specified semantic properties and that can be used as the basis for verifiable model composition.

6. Related Work

Work on AOM can be roughly partitioned into two categories based on the primary focus of the work: Those that provide techniques for modeling aspect-oriented programming (AOP) concepts [11], and those that provide requirements and design modeling techniques that tackle the problem of isolating features in modeling views and studying their interactions. Work in the first category focus on modeling AOP concepts such as join points and advise using either lightweight or heavyweight extensions of modeling languages such as the UML (e.g., see [12, 13, 20, 19]). Metamodels are used in these approaches to describe static concepts. None of the metamodels we have encountered in this area of work describe the dynamics of model composition. Work in this area has produced implementations of model composers. For example, Cottenier et al. [8] use

metamodels to describe the static aspects of an SDL weaver they developed. Specifically, metamodels describe (1) how their weaver views their aspect models, called *aspect beans*, (2) the elements in SDL that can be used as join points, (3) the generic structure of the connections between aspect models and the base model, and (4) the primitives used to specify weaving strategies. The composition metamodels we propose in this paper differ in that they define model composition behavior that can be executed.

Our work and the work on early aspects [1, 5, 16, 17] and subject-oriented modeling [7, 6] fall into the second category. The distinguishing characteristic of work in this area is that it does not focus on retrofitting language-specific AOP concepts at the modeling level. In this area, the Theme approach is the closest to the AOM approach supported by the composition metamodel. In the Theme approach [2, 7, 6], a design, called a *Theme*, is created for each system requirement. A theme is essentially a design view, and thus the Theme approach is similar to the approach described in this paper. Unlike the Theme approach the aspects in our approach are not necessarily tied to a single requirement. Composition relationships in the Theme approach are used to specify how models are to be composed by identifying overlapping concepts in the themes and specifying how models are integrated. The UML metamodel is extended to support composition relationships and describe well-formedness rules for composition. Two types of integration strategies are used: Override and merge. Override integration is used when existing behavior in a subject needs to be updated to reflect new requirements. Merge integration is used when subjects for different requirements are to be integrated. Operations in related subjects may need to be merged into a unified operation. Reconciliation strategies are used to resolve conflicts between property values of corresponding subject elements. Precedence relationships, transformation functions applied to conflicting elements, explicit specification of reconciled elements, and default values may be used for reconciliation. The work described in this paper goes further in that it proposes a composition metamodel that includes behavior that can be executed to compose models in environments such as Kermeta.

Brito and Moreira describe an aspect composition process that identifies match points in a design element and defines composition rules [4]. Rules use identified match points, a binary contribution value (either positive or negative) that quantifies the affects on other aspects, and a priority for a given aspect. We describe the possible relationships between aspects as weave-order relationships and override relationships instead of priority and dependency as done by Brito and Moreira. In addition, our approach uses composition directives to address model mismatch problems.

7. Conclusion

The work presented in this paper demonstrates the viability of using metamodels and metamodeling environments such as Kermeta as the basis for developing tools that compose and otherwise manipulate models. Our experience indicates that implementing composition mechanisms in Kermeta requires less effort than implementing the mechanisms directly in a programming language such as Java. This is because Kermeta shields the developer from some of the complexities associated with processing models, and one can leverage Kermeta's ability to execute behavior defined in metamodels when composing models.

The metamodeling approach described in this paper can be used to create and implement composition metamodels that support other symmetric AOM approaches (e.g., see [6]).

Our future work in this area will focus on developing the next-generation of AOM techniques that will leverage the Kermeta capabilities to support verifiable model composition and to support automated identification of model incompatibilities and generation of composition directives that address the incompatibilities.

Acknowledgements. This material is based upon work partially funded by AFOSR under Award No. FA9550-04-1-0102.

References

[1] Early Aspects Portal. *URL* `http://www.early-aspects.net`, 2006.

[2] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *Proceedings of the International Conference on Software Engineering*, pages 158–167, 2004.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, Vol. 1, MITRE Corporation, March 1973.

[4] I. Brito and A. Moreira. Towards a composition process for aspect-oriented requirements. In *Proceedings of the Early-Aspects Workshop at AOSD2002*, 2002.

[5] R. Chitchyan, A. Rashid, P. Sawyer, A. Garcia, M. Alarcon, J. Bakker, B. Tekinerdogan, S. Clarke, and A. Jackson. Survey of aspect-oriented analysis and design approaches. Technical Report ULANC-9, AOSD - Europe, May 2005.

[6] S. Clarke. "Extending Standard UML with Model Composition Semantics". *Science of Computer Programming*, 44(1):71–100, July 2002.

[7] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *The 23rd International Conference on Software Engineering (ICSE), Toronto, Canada*, 2001.

[8] T. Cottenier, A. V. D. Berg, and T. Elrad. Modeling aspect-oriented compositions. In *Proceedings of the 7th International Workshop on Aspect-Oriented Modeling, in conjunction of the 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS'05)*, Montego Bay, Jamaica, October 2005.

[9] R. B. France, I. Ray, G. Georg, and S. Ghosh. An aspect-oriented approach to design modeling. *IEE Proceedings - Software, Special Issue on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, 151(4):173–185, August 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvaskyla, Finland, June 1997.

[12] J. Kienzle, Y. Yu, and J. Xiong. On composition and reuse of aspects. In *Proceedings of the Foundations of Aspect-Oriented Languages Workshop*, Boston, MA, USA, March 2003.

[13] J. K. M. Kande and A. Strohmeier. From aop to uml - a bottom-up approach. In *Aspect Oriented Modeling workshop held with Aspect Oriented Software Development conference*, Enschede, The Netherlands, April 2002.

[14] OMG Adopted Specification ptc/03-10-04. The Meta Object Facility (MOF) Core Specification. Version 2.0, OMG, http://www.omg.org.

[15] P.Muller, F.Fleury, and J.Jezequel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, pages 264–278, Montego Bay, Jamaica, October 2005.

[16] A. Rashid, A. Moreira, and J. Araujo. Modularization and composition of aspectual requirements. In *2nd International Conference on Aspect-Oriented Software Development*, pages 11–20, Boston, March 2003. ACM.

[17] A. Rashid, P. Sawyer, A. Moreira, and J. Araujo. Early aspects: A model for aspect-oriented requirements engineering. In *IEEE Joint International Conference on Requirements Engineering*, pages 199–202, Essen, Germany, September 2002.

[18] Y. R. Reddy, S. Ghosh, R. France, G. Straw, J. Bieman, N. McEachen, E. Song, and G. Georg. Directives for composing aspect-oriented design class models. *in The Transactions on Aspect-Oriented Software Development*, 2006.

[19] D. Stein, S. Hanenberg, and R. Unland. A UML-based Aspect-Oriented Design Notation For AspectJ. In *Proceedings of the 1st International Conf. on Aspect-oriented software development*, pages 106–112, Enschede, The Netherlands, 2002. ACM Press.

[20] D. Stein, S. Hanenberg, and R. Unland. On representing join points in the uml. In *Aspect Oriented Modeling workshop held with UML 2002*, Dresden, Germany, October 2002.

[21] The Object Management Group. UML 2.0: Superstructure Specification. Version 2.0, OMG, formal/05-07-04, 2005.

[22] TRISKELL. The KerMeta Project Home Page. *URL* `http://www.kermeta.org`, 2005.